AD-A067 546 NEW YORK UNIV N Y COURANT INST OF MATHEMATICAL SCIENCES F/G 9/2
AUTOMATIC DISCOVERY OF HEURISTICS FOR NON-DETERMINISTIC PROGRAM--ETC(U)
JAN 79 S J STOLFO, M C HARRISON N00014-75-C-0571
UNCLASSIFIED TR-007 NL

1 OF 1
ADA
067546

END
DATE
FILMED
6-79
DDC

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

ACCESSION for
NTIS White Section ✓
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

LEVEL II

12

AD A067546

DDC FILE COPY

AUTOMATIC DISCOVERY OF HEURISTICS FOR NON-DETERMINISTIC PROGRAMS

By

SALVATORE J. STOLFO
MALCOLM C. HARRISON

January 1979

Report No. 007

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

"This material is based upon work supported by the Office of Naval Research under Contract No. N00014-75-C-0571, (NR049-347)."

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

79 04 16 081

DDC
RECEIVED
APR 19 1979
D

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 007 (14) TR-~~007~~ 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Automatic Discovery of Heuristics for Non-deterministic Programs.	5. TYPE OF REPORT & PERIOD COVERED (9) Technical repts
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Salvatore J. Stolfo, Malcolm C. Harrison	8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0571 (NR049-347)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Courant Institute of Mathematical Sciences, New York University 099 950, 251 Mercer Street, New York, N.Y. 10012	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research	12. REPORT DATE (11) Jan~~uary~~ 1979
	13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Department of the Navy, Arlington, Virginia 22217	15. SECURITY CLASS. (of this report) unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

(12) 35 p.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

none

18. SUPPLEMENTARY NOTES

none

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Heuristics, Non-deterministic program, Pattern Recognition, Production System, Trace Sequence

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

During the last few years a number of relatively effective AI programs have been written incorporating considerable amounts of problem specific knowledge. Consequently, the problem of encoding such knowledge in a useful form has emerged as one of the central problems of AI. In particular, Declarative representations of knowledge have attracted much attention partly because of the relative ease with which knowledge can be communicated in this

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

099 950

form. Unfortunately, implementation of Declaratively speci-fied knowledge corresponds to a non-deterministic program which incurs enormous computational costs.

This paper discusses one way to limit this cost. The approach we take is to develop control heuristics for a family of problems from traces of sample solutions generated during a training session with a human expert. Algorithms have been developed which recognize a predefined set of patterns in the sequence of 'knowledge applications' and which compile descriptions of these patterns in a control language, called CRAPS. More specifically, patterns of repeating, parallel and common sequences are considered in the analysis. The CRAPS descriptions generated are then used for guidance in solving subsequent problems. We discuss the utility of such an approach and give an example of a generated CRAPS description.

Automatic Discovery of Heuristics

For Non-deterministic Programs

Topic: Induction, Learning and Knowledge Acquisition

Keywords: Heuristics, Non-deterministic program, Pattern Recognition, Production System, Trace Sequence

Authors: Salvatore J. Stolfo
and
Malcolm C. Harrison

Computer Science Department
Courant Institute
New York University
251 Mercer Street
New York, N.Y. 10012

Abstract

During the last few years a number of relatively effective AI programs have been written incorporating considerable amounts of problem specific knowledge. Consequently, the problem of encoding such knowledge in a useful form has emerged as one of the central problems of AI. In particular, Declarative representations of knowledge have attracted much attention partly because of the relative ease with which knowledge can be communicated in this form. Unfortunately, implementation of Declaratively specified knowledge corresponds to a non-deterministic program which incurs enormous computational costs.

This paper discusses one way to limit this cost. The approach we take is to develop control heuristics for a family of problems from traces of sample solutions generated during a training session with a human expert. Algorithms have been developed which recognize a predefined set of patterns in the sequence of 'knowledge applications' and which compile descriptions of these patterns in a control language, called CRAPS. More specifically, patterns of repeating, parallel and common sequences are considered in the analysis. The CRAPS descriptions generated are then used for guidance in solving subsequent problems. We discuss the utility of such an approach and give an example of a generated CRAPS description.

1. Introduction

During the last few years, a number of relatively effective Artificial Intelligence programs have been written incorporating considerable amounts of knowledge, and the problem of encoding such knowledge in a useful form has emerged as one of the central problems of AI. Winograd [16] distinguishes between declarative information which can be thought of as "knowing what", and procedural information which can be thought of as "knowing how". He describes the underlying problem as that of constructing representations which can take advantage of the decomposability of the declarative representation without sacrificing the interactive possibilities of the procedural representation. Ideally, it should be possible to specify information in a form which does not constrain the way in which it is to be used; unfortunately, a straightforward implementation of such a declarative representation corresponds to a non-deterministic program which makes a relatively blind search through the solution space.

More recently, attention has turned towards mechanisms which facilitate incorporating limited procedural or heuristic information into a primarily declarative framework. For example, Rychener's approach [9] is to build a rational "goal" structure into declarative rule-based systems, while Davis [2] favors a separate set of "meta-rules" specifying control information. In some cases,we might expect that the amount of procedural information is of the same order of magnitude as the amount of declarative information, and so such approaches may be effective. However, in most cases the amount of control information will be very large (e.g. theorem-proving, natural language understanding). Thus the problem of acquiring, debugging and extending control information will become increasingly important.

In general, we might expect that this control information will embody very sophisticated principles requiring data-structures not present in the declarative form of the program, or which are only deducible by the use of considerable intelligence. For example, it is clear that the incorporation of the heuristic principles of evaluation and α-β search into a declarative chess program which specifies only the rules of chess but no strategies for playing will require an analysis which is considerably beyond the ability of present techniques; even the optimization of the parameters in a linear evaluation function involves highly sophisticated processing.

However, we believe that there are a number of important areas in which it might be possible to deduce control information automatically from a declarative program. These include declaratively specified problems:

- for which there exists a relatively simple algorithmic procedure;
- whose performance can be improved in frequently occurring or particularly important special cases;
- in which particular subproblems can be solved by simple algorithmic procedures.

Some work on this problem was reported by Fikes, Hart and Nilsson [4]. In their system, STRIPS, sequences of operator applications used in solving sample problems were stored for possible guidance (or planning) in solving subsequent problems. In this paper we report on a somewhat different approach, which stresses the importance of a deeper analysis of the sequence of operator applications. A similar approach has been adopted by Phillips [8].

2. A Sample Problem

As an example, consider the following problem. Suppose we want to solve a (slightly idealized) jig-saw puzzle, in which each piece has an average color, and four sides described by unique integers (with side i fitting side -i). We consider below a non-deterministic algorithm written in production language form. An informal description of the productions we might use would include:

START-PUZZLE: If there is no puzzle currently being built and you are holding a piece, then you can make this piece the first part of the puzzle, leaving your hand empty. Notice that you continue to look at the piece.

LOOK-AT-PIECE-IN-HEAP: If a piece is in the heap then you can look at it.

PICK-UP-OBJECT-IN-VIEW: If you are looking at an object and holding nothing, then you can pick up the object. Its position changes to being in your hand.

LOOK-AT-OBJECT-IN-HAND: If something is in your hand, then you can look at it.

CLOSE-EYES: If you are looking at something which is not NOTHING then you can look at NOTHING.

OBJECT-IN-HAND-IN-VIEW: This production senses when the object being held is in view without performing any actions.

FIND-COLOR-OF-PIECE: If you are holding a puzzle piece, you can look at it and know its color.

PIECE-HAS-CURRENT-COLOR: This senses when the object being held has the current color you are considering.

FORGET-COLOR-OF-PIECE: If a piece is in view and you know its color, you can look away from it and forget the color.

PIECE-HAS-STRAIGHT-EDGE: This production senses when the piece in view has a straight edge.

PUT-PIECE-DOWN-IN-HEAP: If you are holding a puzzle piece, you can put it in the heap and your hand is empty. Notice that the piece continues to be in view.

HEAP-IS-EMPTY: This senses if the heap is empty.

PIECE-FITS-IN-PUZZLE: This production senses when a puzzle piece that is in view will fit another puzzle piece already in the puzzle.

FIT-PIECE-IN-PUZZLE: If the piece being held and in view fits in the puzzle, the sides of the matching pieces can be joined.

PIECE-PUT-IN-PUZZLE: If you are holding a puzzle piece, which is being placed in the puzzle, you could put it down in the puzzle.

PUZZLE-IS-FINISHED: If all of the pieces are in the puzzle, you can halt.

The complete program is given in Appendix I.

It should be noted that this representation is highly non-deterministic, and if executed would be hyper-exponentially inefficient. It does not even "know" that it will find a solution (i.e. terminate) if it can repeat the production PUT-PIECE-IN-PUZZLE as frequently as possible, so even the crudest goal-subgoal structure is absent. (However, there is no production to remove a piece from the puzzle, which serves as a clue to an intelligent observer). It is even possible for the program to pick up a piece and then put it down immediately without doing anything with it.

The performance of this program could be improved to a tolerable level if the following heuristic was added:

```
. use the following sequence repeatedly
  pick up a piece from the heap;
  insert it in the puzzle if it fits.
```

An alternative heuristic would be:

```
. use the following sequence repeatedly
  locate a piece p in the puzzle with a missing neighbor;
  pick up pieces from the heap until one matches p, and
```

insert it in the puzzle.

Each of these heuristics is very simple, consisting mainly of sequencing rules for the productions, and would seem to be within the range of automatic inference. A further improvement could be made by noting that the procedure for selecting a piece from the heap is inefficient since the same piece may be selected repeatedly (and in a production system such as that of Rychener [10] which prefers to use recently referenced items in working memory, in other respects a reasonable strategy, the search will almost always be ineffective). If we add a few more productions, it becomes possible for the system to search systematically through the heap by constructing a pile from the pieces in the heap, and putting a piece down in a different pile when it has looked at it.

MAKE-A-PILE: If you want to make a pile and you are holding something, put it in the pile and your hand is empty. The object continues to be in view.

PICK-A-PILE: If you are not working with a pile, just pick the first one.

PICK-OBJECT-FROM-PILE: To pick an object from a pile, if your hand is empty reach in and pick up the first one you see.

PUT-OBJECT-IN-PILE: To put an object in a pile, if you are holding something, just place it in the back of the pile and your hand is empty.

FORGET-THIS-PILE: To forget a pile, just push the current pile behind all the others.

PILE-IS-EMPTY: A pile is empty, if there is nothing on it.

DESTROY-A-PILE: To destroy the current pile, just strike it from memory.

THERE-ARE-NO-PILES: There are no piles if all the piles were destroyed.

LOOK-AT-FIRST-IN-PILE: If there is an object at the front of the pile, you can focus on it.

LOOK-AT-NEXT-IN-PILE: If you want to scan through a pile and you are looking at the first object in it, focus on the next one by placing the first object behind the last.

The actual pile productions can be found in Appendix II.

The pile productions are very general, and might be expected to be present in any system; indeed, it seems likely that productions for dealing with heaps and piles (i.e. sets and tuples) and heuristics for implementing important operations such as searching will be present in production systems as they are in modern programming languages [3].

Carrying our jig-saw puzzle a step or two further, sufficient productions are present now to permit simple sequencing heuristics to reflect the usual strategies used by experienced puzzle-solvers:

. work on the outside edges first (build a pile of outside edges); repeat the PUT-PIECE-IN-PUZZLE sequence until this pile is empty;

. separate the pieces in the heap into piles of the same average color, and search the appropriate pile first;

. search first for pieces which have more than one neighbor of the same average color (e.g. work on the sky first);

It may appear that automatic detection of such heuristics is a task of great difficulty. Below we outline an approach to this problem which we believe can be quite successful, describe a system which has been embeeded in PROSYS, an extension of OPS2 [5], and give some preliminary results.

3. Approach

Our procedure is as follows: we select a 'typical' input to the program and run the program repeatedly on this input, recording the sequence of rules selected. This is repeated for other typical inputs. We then attempt to describe the better (i.e. shorter) successful sequences in a language, CRAPS, designed for this purpose and described in the next section. We then use this description to guide the program's subsequent decisions.

Inherent in this approach is the assumption that good decision-making procedures or heuristics can be inferred from the performance of the program on only selected inputs. In general, as suggested by work on inductive inference [17] and information theory [1,6,11], we will give preference to short CRAPS descriptions which will generate a high proportion of short successful solutions and few long or unsuccessful solutions. We anticipate that the selection of inputs will be critical and that eventually we will want to be able to handle new inputs incrementally, as was done in [17]. Initially, however, we will concentrate on the simpler problem of getting a good solution for the non-incremental case.

As suggested by the example in section 2, the execution-time of a completely declarative program will usually be too long to permit a solution except in the simpler cases. Accordingly, we will run the program in "training mode" in which its decisions can be observed and if necessary corrected. In initial experiments this will be done by a trainer who is aware of the structure of the program; subsequently we anticipate the necessity of using techniques similar to those of Davis [2], which will enable the trainer to deal only with the external behavior of the program.

It should be noted that we have deliberately chosen to exclude information about sequences which end in failure. It is clear that, as found by Winston in other inductive inference work [17], counter-examples will be extremely valuable. However, as the reader will note below, even this simpler problem poses considerable technical difficulties, and it was our feeling that a clearer picture would emerge from the simpler approach. Furthermore, initial experiments described in section 5 suggest that useful results can be obtained without counter-examples.

4. The CRAPS language

The CRAPS language provides a semantic framework with which to specify or describe sequences of rule applications in the execution of the non-deterministic program. The most basic primitive of CRAPS is called a unit. A unit specifies either a rule application with preconditions, or a control operation applied to a sequence of units. The control operations are:

(1) Concatenation of units producing sequences;

(2) Repetition of a sequence of units controlled by Boolean conditions;

(3) Alternative (or conditional) selection of a sequence from a set of sequences;

(4) Permutation of a set of sequences. (Thus, the acronym CRAPS.)

Conditions in the above are DNF expressions of rules, with a rule being considered true if it is fireable.

The operators of CRAPS correspond to control primitives of conventional programming languages. Concatenation corresponds to sequential execution of statements, and repetition corresponds to iteration statements with 'while' and 'until' termination conditions. The alternation operator specifies alternative sequences of actions much like an ALGOL-68 'case' statement or LISP 'cond' expression. The permutation primitive represents a form of concurrent execution similar to the specification of collateral expressions in ALGOL-68.

An example of a CRAPS description can be found in section 5.

5. An Example: Binary Tree Traversal

The program in figure 1 has the ability to scan a binary tree in any order. No control information is specified in any of the productions. The productions have been chosen to encode all of the relevant information about scanning a binary tree without specifying direction.

A binary tree is represented in working memory by the following data format. The root of the tree is represented by (ROOT =X) where =X can be bound to any symbol. If node B has a left son A, it is represented by (LEFT B A) and similarly (RIGHT B C) represents C as the right son of B. The father of a node (either left or right) is represented for example by (FATHER B A). The current node being scanned is represented by (NODE =X). In scanning binary trees, nodes can be printed only once. If node A has been scanned and printed, then (ALREADY-P A) is deposited in working memory.

Several binary trees were placed in working memory and then the program was initiated and directed by a human to scan in-order each tree. A trace for each execution was produced (see figure 2 for examples). The entire set of traces was presented to the pattern recognition algorithms, which produced the CRAPS description presented in figure 3 . The integers labelling the rule names in the final description are meant to differentiate particular instances of rule applications. For example (15.PRINT) is a completely different application of PRINT than is (17.PRINT).

In this particular case the CRAPS description is a precise definition of an in-order balanced binary tree scan program. That is, given a working memory specifying a balanced binary tree, the shortest execution sequence of the program which would result in an in-order print of the nodes would belong to the set of sequences specified by the CRAPS description; furthermore, the CRAPS description provides enough information to determine which production should be fired at every point in the execution of the program. Thus a non-deterministic program has in effect been reduced to a deterministic program.

In general, we cannot assume that the CRAPS description will be as effective as this. With poor training sequences, or incorrect analyses of execution traces, CRAPS descriptions may imply heuristics which are not helpful, or which are helpful for some examples and harmful in others. The initial approach we implemented was that of Rychener's. The existing program was altered by including special 'tag' or 'control elements' in both sides of the rules to reflect the control information in the description. This approach appeared to impose too high a level of control affecting much of the quality of the production system representation which makes it attractive. Although the original trees used in the training sequence and other trees were able to be traversed correctly, several other instances of trees were not. This has led us to consider Davis' approach of "meta-rules" with a probabilistic component to them. In this direction, a CRAPS description would be transformed to a set of rules which suggest preferences of rules over others during the testing and selection process on each execution cycle. We are currently looking at ways of doing this.

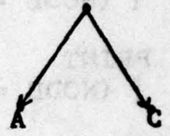
Figure 1, Binary Tree Scanning Production System

```
START-POINTING
[ (ROOT =x)  -(NODE  =anything)  -->  (NODE =x) ]

GO-LEFT
[(NODE =x)  (LEFT =x =y)  -->  (<delete> (NODE =x)) (NODE =x) (FATHER =x =y) ]

GO-RIGHT
[ (NODE =x)  (RIGHT =x =y) -->  (<delete> (NODE =x)) (NODE =y) (FATHER =x =y) ]

CAN-T-GO-LEFT
[ (NODE =x) -(LEFT =x =anything) -->  ]

CAN-T-GO-RIGHT
[ (NODE =x) -(RIGHT =x =anything) -->  ]

PRINT
[ (NODE =x) -(ALREADY-P =x)  --> (<write> =x) (ALREADY-P =x) ]

GO-UP
[ (NODE =x) (FATHER =x =y)  -->  (<delete> (NODE =x)) (NODE =y) ]

STOP
[ (NODE =x) (ROOT =x)  -->  (<delete> (NODE =x)) (<halt>) ]
```

FIGURE 2. SAMPLE TRACES FROM BINARY TREE PS RUN.

```
<<
(START-POINTING NIL (1) (2 2))
(PRINT (STOP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (1 2) (3 4))
(STOP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (1 4) (5 6))
>>
```

```
<<
(START-POINTING NIL (9) (10 10))
(GO-LEFT (STOP PRINT GO-RIGHT) (9 10) (11 13))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (12 13) (14 15))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (12 15) (16 17))
(PRINT (STOP GO-LEFT GO-RIGHT) (9 17) (18 19))
(GO-RIGHT (STOP GO-LEFT) (9 19) (20 22))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (21 22) (23 24))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (21 24) (25 26))
(STOP (GO-LEFT GO-RIGHT) (9 26) (27 28))
>>
```



```
<<
(START-POINTING NIL (14) (15 15))
(GO-LEFT (STOP PRINT GO-RIGHT) (14 15) (16 18))
(GO-LEFT (GO-UP PRINT GO-RIGHT) (17 18) (19 21))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (20 21) (22 23))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (20 23) (24 25))
(PRINT (GO-UP GO-LEFT GO-RIGHT) (17 25) (26 27))
(GO-RIGHT (GO-UP GO-LEFT) (17 27) (28 30))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (29 30) (31 32))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (29 32) (33 34))
(GO-UP (GO-LEFT GO-RIGHT) (17 34) (35 36))
(PRINT (STOP GO-LEFT GO-RIGHT) (14 36) (37 38))
(GO-RIGHT (STOP GO-LEFT) (14 38) (39 41))
(GO-LEFT (GO-UP PRINT GO-RIGHT) (40 41) (42 44))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (43 44) (45 46))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (43 46) (47 48))
(PRINT (GO-UP GO-LEFT GO-RIGHT) (40 48) (49 50))
(GO-RIGHT (GO-UP GO-LEFT) (40 50) (51 53))
(PRINT (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) (52 53) (54 55))
(GO-UP (CAN-T-GO-RIGHT CAN-T-GO-LEFT) (52 55) (56 57))
(GO-UP (GO-LEFT GO-RIGHT) (40 57) (58 59))
(STOP (GO-LEFT GO-RIGHT) (14 59) (60 61))
>>
```

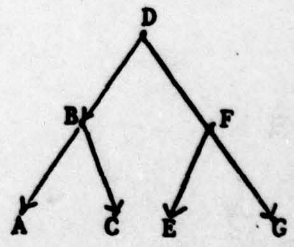


FIGURE 3. SAMPLE CRAPS OUTPUT FROM BINARY TREE PS RUN.

```
<<
 ((0 . START-POINTING) NIL)
 [IF (GO-LEFT STOP PRINT GO-RIGHT) THEN
      <<
       [REPEAT   (WHILE  (GO-LEFT GO-RIGHT))
                 (UNTIL  NIL)
           <<
             [REPEAT   (WHILE  (GO-LEFT GO-RIGHT))
                       (UNTIL  (CAN-T-GO-LEFT CAN-T-GO-RIGHT))
                 <<
                  ((13 . GO-LEFT) (STOP PRINT GO-RIGHT))
                 >>
             ]
             [REPEAT   (WHILE  (CAN-T-GO-LEFT CAN-T-GO-RIGHT))
                       (UNTIL  (GO-LEFT GO-RIGHT))
                 <<
                  ((15 . PRINT) (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT))
                  [REPEAT    (WHILE  (GO-UP))
                             (UNTIL  (PRINT STOP))
                       <<
                        ((16 . GO-UP) (CAN-T-GO-RIGHT CAN-T-GO-LEFT))
                       >>
                  ]
                  ((17 . PRINT) (GO-UP GO-LEFT GO-RIGHT))
                  ((18 . GO-RIGHT) (GO-UP GO-LEFT))
                 >>
             ]
           >>
       ]
      >>
  ELSEIF TRUE THEN
      <<
      >>
 ]
 ((1 . PRINT) (STOP CAN-T-GO-RIGHT CAN-T-GO-LEFT))
 [IF (GO-UP CAN-T-GO-RIGHT CAN-T-GO-LEFT) THEN
      <<
       [REPEAT   (WHILE  (GO-UP))
                 (UNTIL  (STOP))
           <<
             ((30 . GO-UP) (CAN-T-GO-RIGHT CAN-T-GO-LEFT))
           >>
       ]
      >>
  ELSEIF TRUE THEN
      <<
      >>
 ]
 ((2 . STOP) (CAN-T-GO-RIGHT CAN-T-GO-LEFT))
>>
```

6. Algorithms

In this section we outline the algorithms which produce CRAPS descriptions from a set of sequences of rule applications which solved a particular family of problems. The exact form of an input sequence is << P1 P2 .... >>, where Pi is of the form $(P_i' \; (P_{i1}P_{i2}...P_{im}) (n_{i1}n_{i2}...n_{ij}) \; (o_{i1}o_{i2}))$, where P_i' is the rule which was applied on the i'th cycle of execution. $(P_{i1}P_{i2}..$ $.P_{im})$ is the list of rules which were applicable on the ith cycle and describes the data environment at that point in the execution. $(n_{i1}n_{i2}...n_{ij})$ is the list of unique numbers associated with the instances of data elements which matched the rule P_i', and $(o_{i1}o_{i2})$ is the range of numbers associated with the data elements deposited in working memory by the application of rule P_i'. The sequence presented is an exact history of the execution of the program and is therefore already partially ordered by the back dominance relation. This makes the subsequent pattern recongnizing easier.

The initial approach we considered to finding permutations of rule applications in the sequences was to do a static analysis of the rules to determine which rules were permutable, i.e., which rules produced output that could not be input by other rules. This approach was found to be too difficult and not general enough. Instead, we use techniques similar to those developed for program optimization. Using the input/output information contained in the trace, a data flow directed acyclic graph (DAG) is constructed whose nodes correspond to rule applications and whose edges represent the flow of data from one rule to another. This DAG completely specifies all of the data dependencies and therefore all of the possible execution sequences. After the DAG is cleansed of irrelevant forward arcs (see Tarjan [13,14,15]), it is linearized to a CRAPS description employing permutation and simple units only. The construction of permutations is based on the observation that if a node has two or more entering edges then the subsequences associated with its predecessors can be permuted. A simple sequential scan of the nodes in the DAG according to the partial ordering is sufficient to produce a linear description in linear time.

The next processing is the detection of repeating subsequences and formation of repetition units. Unlike the permutation detection problem, which is well-defined, and which is solved by the algorithm described above, some sequences may have several equally acceptable descriptions using the repetition operator. The algorithm we use gives one description. We first scan the sequence identifying all unique units. All such units cannot be part of a repetition and thus divide the sequence into shorter subsequences to which the entire procedure can be applied recursively. Once a candidate subsequence is found (every unit in it appears at least twice), we scan from left to right repeatedly looking for equivalent adjacent subsequences replacing them by repetition units. The procedure stops when it considers lengths greater than one half the current length of the sequence.

The final operation to be done is the coalescing of several descriptions produced by the above processing into one description employing the alternation operator. The construction we consider is similar to that of Winston and Hayes-Roth. From two or more descriptions of some object we identify the points of similarity and alternate the differences. Since this problem is equivalent to the Longest Common Subsequence problem, which is known to be NP-Complete (See Maier [7]), our approach is highly heuristic. The computation of a common subsequence is performed by a concurrent left to right scan of all sequences with a heuristic weighting function which gives preference to repitition units over permutation units and simple rule application units.

The conditions for repetition and alternation units are computed by simple set expressions on the list of applicable rules ($(P_{i1}P_{i2}...P_{im})$ above) at the time a set of subsequences are collapsed into a unit. The final output of these algorithms is fed back to the production system interpreter and used for further problem solutions. Complete details of these algorithms can be found in Stolfo [12].

7. Conclusion

It is believed by many researchers that an important quality of intelligent behavior is the ability to improve performance with experience, and that generalizing a concept is a critical aspect of learning. In CRAPS, a form of generalization occurs when a repeating subsequence is collapsed into a repetition unit. Although quite restricted in scope, CRAPS shows how a system might learn procedures, a form of learning which we believe is very important. Although there are a number of very difficult technical problems, it seems to us that with more powerful pattern recognition techniques and more powerful generalizations of control statements, this approach could be very fruitful.

With less ambitious designs, CRAPS can be viewed as a programming aid for the designer and implementer of a large AI problem-solving knowledge base. The CRAPS approach might be useful in fine-tuning a declarative knowledge base as opposed to 'hand-compiling' control elements to effect competent performance in such a system.

The power of the descriptions produced is limited by both the expressive power of the CRAPS primitives, and the level of sophistication of the pattern recognition algorithms that have been developed. For example, during repetition detection no notion of similar subsequence is used; rather, a subsequence must appear exactly, which prevents alternation units from appearing within repetitions. Despite this, the algorithms are powerful enough to detect interesting patterns and subsequently interesting heuristics.

Our experiments suggest that this approach will have the best chance of success when the encoding of the knowledge of the problem domain is such that on any execution cycle a small number of productions and only one instantiation of each production is applicable. This suggests one way in which CRAPS should evolve: the execution traces should include not only productions but also the instantiations of productions. However, recognizing patterns in the sequences which include data is a much more difficult problem. We view our approach as an initial step in the understanding of this more general problem.

APPENDIX I

Jigsaw Puzzle Production System

In what follows, data elements can be any LISP data structure. An atomic data element must match exactly and a list must match a list with the same structure and content. A symbol preceded with an equals sign (=) represents a variable which can match any data structure. The $ symbol contained in productions has the same function as the SNOBOL4 conditional assignment operator (.). When a rule is fired, the data elements are not deleted from working memory unless they are included as arguments to the <delete> system function in the right hand side of the rule. The other system functions contained in the rules are represented in lower case and enclosed by pointed brackets (< >). Their function is described by their names. The symbol ! is an operator which matches the entire remaining portion of the list following it. Where it appears in the right hand side, it deposits the list that matched but it strips away the enclosing parentheses.

Working Memory Data Elements:

1. (PIECE p c) represents a puzzle piece where p is a unique integer associated with a piece and c is its average color. We will assume the machine knows at any time where piece p is located and so we ignore location in the representation.

2. (SIDE s p n c) represents the side of a puzzle piece. s is either LEFT, RIGHT, TOP or BOTTOM of piece p. Since location is ignored, rotation of a piece in space is also ignored. n is an integer representing the shape of the side where +n mates with -n and 0 represents a straight edge. c is the color of the edge. The representation could easily be extended to represent a vector of colors for a series of points on the edge.

3. (NUMBER-OF-PIECES n) states that n is the total number of puzzle pieces.

4. (JOINED (side1 side2) (side3 side4)...) represents the sequence of sides that were joined in forming the puzzle. The symbol sidei is of the form 2 above.

5. (IN-PUZZLE (PIECE p c)) states that piece p is in the puzzle.

6. (NUMBER-IN-PUZZLE n) states that n is the number of puzzle pieces that are in the puzzle.

7. (CURRENT-COLOR c) represents the current color of an object being considered.

8. (HOLDING p) states that p is the current object being held (for example a puzzle piece). If the hand is empty, (HOLDING NOTHING) is in working memory.

9. (IN-HEAP (PIECE p c)) represents puzzle piece p as being in a heap with no implicit ordering of edges.

10. (LOOKING-AT x) states that object x is in view. If nothing is in view then (LOOKING-AT NOTHING) is in working memory.

11. (BEING-PUT-IN-PUZZLE x) represents the current piece x that is actively being placed in the puzzle.

Production Memory:

```
    LOOK-AT-PIECE-IN-HEAP
    [(LOOKING-AT =object) $ =c1
     (IN-HEAP (PIECE =p =c) $ =object)
                                          -->
                                              (<delete> =c1)
                                              (LOOKING-AT =object)]

    LOOK-AT-OBJECT-IN-HAND
    [(LOOKING-AT =something) $ =c1
     (HOLDING =object)
                                          -->
                                             (<delete> =c1)
                                             (LOOKING-AT =object)]

    CLOSE-EYES
    [(LOOKING-AT =something) $ =c1
    -(LOOKING-AT NOTHING)
                                          -->
                                             (<delete> =c1)
                                             (LOOKING-AT NOTHING)]

    OBJECT-IN-HAND-IN-VIEW
    [(HOLDING =object)
     (LOOKING-AT =object)
    -(LOOKING-AT NOTHING)
                                          -->
                                             ]

    PICK-UP-OBJECT-IN-VIEW
    [(HOLDING NOTHING)
     (LOOKING-AT =object)
     (IN-HEAP =object) $ =c3
                                          -->
                                             (<delete> =c3)
                                             (LOOKING-AT =object)
                                             (HOLDING =object)]
```

```
PUT-PIECE-DOWN-IN-HEAP
[(HOLDING (PIECE =p =c)) $ =c1
 (LOOKING-AT =object)
                              -->
                                 (<delete> =c1)
                                 (LOOKING-AT =object)
                                 (HOLDING NOTHING)
                                 (IN-HEAP (PIECE =p =c))]

FIND-COLOR-OF-PIECE
[(LOOKING-AT (PIECE =p =c))
 (CURRENT-COLOR =anything) $ =c2
                              -->
                                 (<delete> =c2)
                                 (CURRENT-COLOR =c)]

FORGET-COLOR-OF-PIECE
[(LOOKING-AT (PIECE =p =c)) $ =c1
 (CURRENT-COLOR =c) $ =c2
                              -->
                                 (<delete> =c1 =c2)
                                 (LOOKING-AT NOTHING)
                                 (CURRENT-COLOR NOTHING)]

PIECE-HAS-STRAIGHT-EDGE
[(LOOKING-AT (PIECE =p =c))
 (SIDE =any =p 0 =c)
                             -->
                                ]

PIECE-HAS-CURRENT-COLOR
[(LOOKING-AT (PIECE =p =c))
 (CURRENT-COLOR =c)
                             -->
                                ]

HEAP-IS-EMPTY
[-(IN-HEAP =anything)
                             -->
                                ]

START-PUZZLE
[-(IN-PUZZLE =anything)
 (HOLDING (PIECE =p =c) $ =object) $ =c2
 (LOOKING-AT =object)
                            -->(JOINED )
                               (<delete> =c2)
                               (HOLDING NOTHING)
                               (IN-PUZZLE =object)
                               (LOOKING-AT =object)
                               (NUMBER-IN-PUZZLE 1)]

PIECE-FITS-IN-PUZZLE
[(LOOKING-AT (PIECE =p =c) $ =object)
-(IN-PUZZLE =object)
 (IN-PUZZLE (PIECE =p2 =otherc) $ =otherp)
 (SIDE =any =p =n =c)
 (SIDE =another =p2 (<negative> =n) =k)
                            -->  ]
```

```
FIT-PIECE-IN-PUZZLE
[(HOLDING (PIECE =p =c) $ =object)
-(IN-PUZZLE =object)
 (LOOKING-AT =object)
 (IN-PUZZLE (PIECE =p2 =otherc) $ =otherp)
 (SIDE =any =p =n =c) $ =c4
 (SIDE =another =p2 (<negative> =n) =k) $ =c5
 (JOINED ! =rest) $ =c6
                                            -->
                                               (<delete> =c4 =c5 =c6)
                                               (BEING-PUT-IN-PUZZLE =object)
                                               (LOOKING-AT =object)
                                               (JOINED ! =rest (=c4 =c5))]

PIECE-PUT-IN-PUZZLE
[(HOLDING (PIECE =p =c) $ =object) $ =c1
 (NUMBER-IN-PUZZLE =m) $ =c2
-(IN-PUZZLE =object)
 (BEING-PUT-IN-PUZZLE =object) $ =c4
                                            -->
                                               (<delete> =c1 =c2 =c4)
                                               (HOLDING NOTHING)
                                               (LOOKING-AT NOTHING)
                                               (IN-PUZZLE =object)
                                               (NUMBER-IN-PUZZLE (<add1> =m))]

PUZZLE-IS-FINISHED
[(NUMBER-IN-PUZZLE =n)
 (NUMBER-OF-PIECES =n)
-(IN-HEAP =anything)
                                            -->
                                               (<halt>)]
```

APPENDIX II
===========

Extensions to Jigsaw Puzzle Production System

Working Memory:

1. (NUMBER-OF-PILES m) m is the number of piles we have (initially 0).

2. (CURRENT-PILE n) represents the current pile we are using.

3. (ALL-PILES p1 p2 ...) represents all of the piles we've made where pi is a unique integer associated with a pile.

4. (PILE i t1 t2 ...) represents the contents of pile i where tj can be any object.

Production Memory:

```
    MAKE-A-PILE
    [(HOLDING (<not> NOTHING) $ =object) $ =c1
     (LOOKING-AT =object)
     (NUMBER-OF-PILES =m) $ =c3
     (ALL-PILES ! =r) $ =c4
     (CURRENT-PILE NONE) $ =c5
                                          -->
                                             (<delete> =c1 =c3 =c4 =c5)
                                             (NUMBER-OF-PILES (<add1> =m))
                                             (LOOKING-AT =object)
                                             (ALL-PILES (<add1> =m) ! =r)
                                             (CURRENT-PILE (<add1> =m))
                                             (PILE (<add1> =m) =object)
                                             (HOLDING NOTHING)]

    PICK-A-PILE
    [(CURRENT-PILE NONE) $ =c1
     (ALL-PILES =p1 ! =r)
                                          -->
                                             (<delete> =c1)
                                             (ALL-PILES =p1 ! =r)
                                             (CURRENT-PILE =p1)]

    PICK-OBJECT-FROM-PILE
    [(CURRENT-PILE =p1)
     (HOLDING NOTHING) $=c2
     (ALL-PILES =p1 ! =r)
     (PILE =p1 =object ! =rest) $ =c4
     (LOOKING-AT =object)
                                          -->
                                             (<delete> =c2 =c4)
                                             (LOOKING-AT =object)
                                             (HOLDING =object)
                                             (PILE =p1 ! =rest)]
```

30

```
PUT-OBJECT-IN-PILE
[(CURRENT-PILE =p1)
 (HOLDING (<not> NOTHING) $ =object) $ =c2
 (LOOKING-AT =object)
 (ALL-PILES =p1 ! =r)
 (PILE =p1 ! =rest) $ =c5
                                  -->
                                      (<delete> =c2 =c5)
                                      (LOOKING-AT =object)
                                      (HOLDING NOTHING)
                                      (PILE =p1 ! =rest =object)]

FORGET-THIS-PILE
[(CURRENT-PILE =p1) $ =c1
 (ALL-PILES =p1 ! =rest) $ =c2
                                  -->
                                      (<delete> =c1 =c2)
                                      (CURRENT-PILE NONE)
                                      (ALL-PILES ! =rest =p1)

PILE-IS-EMPTY
[(CURRENT-PILE =p1)
 (ALL-PILES =p1 ! =rest)
 (PILE =p1)
                                  -->
                                      ]

DESTROY-A-PILE
[(CURRENT-PILE =p1) $ =c1
 (ALL-PILES =p1 ! =rest) $ =c2
 (PILE =p1 ! =r) $ =c3
                                  -->
                                      (<delete> =c1 =c2 =c3)
                                      (CURRENT-PILE NONE)
                                      (ALL-PILES ! =rest)]

THERE-ARE-NO-PILES
[(CURRENT-PILE NONE)
 (ALL-PILES)
                                  -->
                                      ]

LOOK-AT-FIRST-IN-PILE
[(LOOKING-AT =anything) $ =c1
 (CURRENT-PILE =p1)
 (PILE =p1 =object ! =rest)
                                  -->
                                      (<delete> =c1)
                                      (LOOKING-AT =object)]

LOOK-AT-NEXT-IN-PILE
[(LOOKING-AT =object) $ =c1
 (CURRENT-PILE =p1)
 (PILE =p1 =object =next ! =r) $ =c3
                                  -->
                                      (<delete> =c1 =c3)
                                      (LOOKING-AT =next)
                                      (PILE =p1 =next ! =r =object)]
```

References

[1] G. Chaitin, A Theory of Program Size Formally Identical to Information Theory, J. ACM, Vol 22, 1975.

[2] R. Davis, Applications of Meta-Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases, Ph.D. Thesis, Stanford U., 1976.

[3] R. Dewar, The SETL Programming Language, Courant Institute NYU, 1978, (unpublished preprint).

[4] R. Fikes, P. Hart and N. Nilsson, Learning and Executing Generalized Robot Plans, AI3, 1972.

[5] C. Forgy, J. McDermott, The OPS2 Reference Manual, Department of Computer Science, CMU, 1977.

[6] A. Kolmogorov, Three Approaches to the Quantitative Definition of Information, Problems in Information Transmission 1, 1965.

[7] D. Maier, The Complexity of Some Problems on Subsequences and Supersequences, J. ACM 25-2, 1978.

[8] J. Phillips, Program Inference from Traces using Multiple Knowledge Sources, Proc. IJCAI 5, 1977.

[9] M. Rychener, Control Requirements for the Design of Production System Architectures, Proc. Symp. on AI and Programming Languages, 1977.

[10] M. Rychener, Production Systems as a Programming Language for Artificial Intelligence Research, Ph.D. Thesis, Computer Science, CMU, 1976.

[11] R. Solomonoff, A Formal Theory of Inductive Inference, Information and Control 7, 1964.

[12] S. Stolfo, Automatic Discovery of Heuristics from Sample Execution Traces for Non-deterministic Programs, Ph.D. Thesis, Courant Institute, NYU, 1979 (in preparation).

[13] R. Tarjan, Depth-first Search and Linear Graph Algorithms, SIAM J. Comput., 1972.

[14] R. Tarjan, Finding Dominators in Directed Graphs, SIAM J. Comput., 1972.

[15] R. Tarjan, Testing Flow Graph Reducibility, J. Comput. and Systems Sci., 1974.

[16] T. Winogrod, Frame Representation and the Declarative/ Procedural Controversy, in Representation and Understanding, Bobrow and Collins (ed.), Academic Press, 1975.

[17] P. Winston, Learning Structural Descriptions from Examples, MAC TR-76, MIT, 1976.

Distribution List for Contract No. N00014-75-C-0571

Defense Documentation Center Cameron Station Alexandria, Virginia 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217	2 copies
Office of Naval Research Code 102IP Arlington, Virginia 22217	6 copies
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, Massachusetts 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, Illinois 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, California 91106	1 copy
New York Area Office 715 Broadway - 5th Floor New York, New York 10003	1 copy
Naval Research Laboratory Technical Information Division, Code 2627 Washington, D.C. 20375	6 copies
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps (Code RD-1) Washington, D.C. 20380	1 copy
Office of Naval Research Code 455 Arlington, Virginia 22217	1 copy
Naval Electronics Laboratory Center Advanced Software Technology Division Code 5200 San Diego, California 92152	1 copy
Mr. E.H. Gleissner Naval Ship Research & Development Center Computation and Mathematics Department Bethesda, Maryland 20084	1 copy

Captain Grace M. Hopper NAICOM/MIS Planning Branch (OP-916D) Office of Chief of Naval Operations Washington, D.C. 20350	1 copy
Mr. Kin B. Thompson Technical Director Information Systems Division (OP-91T) Office of Chief of Naval Operations Washington, D.C. 20350	1 copy
Dr. R.T. Chien University of Illinois Coordinated Science Laboratory Urbana, Illinois 61801	1 copy
Office of Naval Research Code 432 Arlington, Virginia 22217	1 copy
Professor John Birk University of Rhode Island Department of Electrical Engineering Kelley Hall Kingston, Rhode Island 02881	1 copy